Strategie di ricerca

Seminario del 15 e 18 ottobre 2018

- *Scopo:*
  1. Assestare la comprensione delle strategie di ricerca viste a lezione
  2. Imparare ad utilizzare la libreria aima.search

- *Com'è organizzata l'esercitazione:*
  1. Velocissimo ripasso delle strategie
  2. Un po' di osservazioni su come rappresentare lo stato
  3. Introduzione alla libreria aima.search
  4. Esempio di utilizzo

Strategie di ricerca

- I problemi si possono modellare come **Problemi di Ricerca** in uno spazio degli stati (**Strategie di Ricerca**).

- Lo spazio degli stati è l'insieme di tutti gli stati raggiungibili dallo stato iniziale con una qualunque sequenza di operatori.

- **Lo Spazio degli Stati è caratterizzato da:**
  - Uno stato iniziale in cui l'agente sa di trovarsi;
  - Un **insieme di azioni** possibili che sono disponibili da parte dell'agente (Operatori che trasformano uno stato in un altro o più formalmente una funzione successore S(X) che riceve in ingresso uno stato e restituisce l'insieme degli stati raggiungibili).
  - Un **cammino** che è una sequenza di azioni che conduce da uno stato a un altro.

TEST: raggiungimento del goal

- La verifica può essere solo l'appartenenza dello stato raggiunto all'insieme dello stato (o degli stati) goal.

- Altri obiettivi (non solo raggiungere il goal, ma...):
  - trovare la sequenza di operatori che arrivano al goal;
  - trovare tutte le soluzioni;
  - trovare una soluzione ottima.
  - In quest'ultimo caso vuol dire che una soluzione può essere preferibile a un'altra.
  - Una funzione costo di cammino assegna un costo a un cammino (in gran parte dei casi quale somma del costo delle azioni individuali lungo il cammino).

Agente risolutore

- L'agente formula un **obiettivo *goal*** e un **problema *problem***
- Quindi **cerca (*SEARCH*)** una **sequenza *seq*** di azioni in grado di risolverlo
- Passa poi ad eseguire le azioni della sequenza una per volta

Agente risolutore

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Cercare soluzioni

Alcuni concetti:

- *Espansione:* si parte da uno stato e applicando gli operatori (o la funzione successore) si generano nuovi stati.

- *Strategia di ricerca:* ad ogni passo scegliere quale stato espandere.

- *Albero di ricerca:* rappresenta l'espansione degli stati a partire dallo stato iniziale (la radice dell'albero). Le foglie dell'albero rappresentano gli stati da espandere.

STRATEGIE DI RICERCA

- STRATEGIE DI RICERCA **NON-INFORMATE**:
  - breadth-first (a costo uniforme);
  - depth-first;
  - depth-first a profondità limitata;
  - ad approfondimento iterativo.

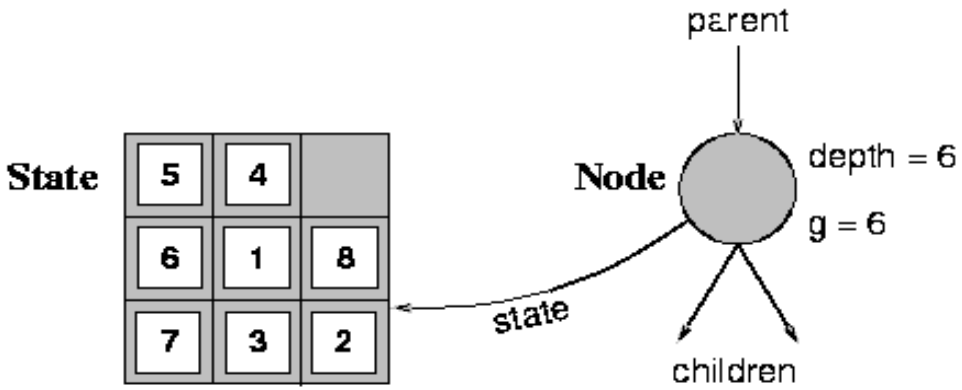
STRATEGIE DI RICERCA

- STRATEGIE DI RICERCA **INFORMATE**:
  - Best first
    1. Greedy
    2. A*
    3. IDA*
    4. SMA*

IDA* e SMA* non sono stati visti a lezione… li potete trovare sul Russell-Norvig nel capitolo dedicato alle ricerche informate.

Strutture dati per l'albero di ricerca (struttura di un nodo)

- Lo stato nello spazio degli stati a cui il nodo corrisponde.
- Il nodo genitore.
- L'operatore che è stato applicato per ottenere il nodo.
- La profondità del nodo (*depth*).
- Il costo del cammino dallo stato iniziale al nodo (*g*).



L'algoritmo generale di ricerca

```
function GENERAL-SEARCH(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

L'algoritmo generale di ricerca

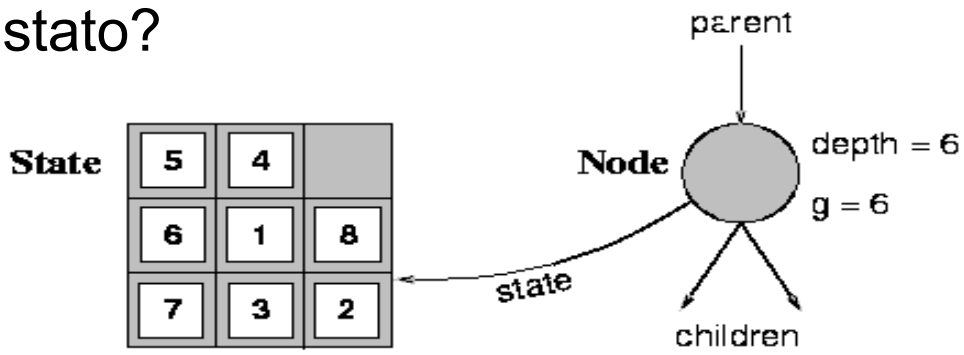
```
function GENERAL-SEARCH(problem, QUEUING-FN) returns a solution, or failure

  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
    if nodes is empty then return failure
    node ← REMOVE-FRONT(nodes)
    if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end
```

Tramite l'argomento **Queuing-Fn** viene passata una funzione per accodare i nodi ottenuti dall'espansione

Primo passo: Definizione del problema

- Come rappresento un problema?
  1. In generale utilizzo una rappresentazione a stati (ho quindi degli **operatori** che mi permettono di operare sugli stati)
  2. Ho uno **stato iniziale**
  3. Ho un **goal** da soddisfare
- Come rappresento uno stato?
  - Strutture dati che rappresentano lo stato
- E gli operatori sullo stato?



Primo passo: Definizione del problema

- Usando un approccio "object-oriented", rappresento uno stato tramite una classe
- Gli operatori sullo stato vengono rappresentati tramite metodi della classe stessa
- E poi, quali altri metodi?
  - boolean `isGoalTest(Object state)` → dice se ho raggiunto il goal (metodo definito dall'interfaccia `aima.search.framework.GoalTest`)
- La libreria `aima` non pone nessun vincolo su come rappresentare lo stato: richiede solo che sia una istanza di `java.lang.Object`.
- **`http://aima.cs.berkeley.edu/java/doc/javadoc/overview-tree.html`**

…operatori sullo stato… bastano questi?

- Con questi metodi, quali strategie posso applicare?
  1. Breadth-first
  2. Depth-first
  3. Depth-bounded
  4. Iterative Deepening

- Ma se non tengo traccia della "strada" percorsa per giungere alla soluzione, posso solo dire che esiste una soluzione ma non posso dire come generarla
  - Es:nel gioco del filetto, so che c'è una soluzione, ma se non conosco le mosse per giungervi…

Il concetto di successore

- Il successore è una struttura dati che tiene traccia di:
  1. stato
  2. operatore applicato per giungere in tale stato
- La libreria **`aima.search.framework`**[1] offre
  - la classe `Successor,` il cui costruttore riceve operatore e stato raggiunto
  - l'interfaccia `SuccessorFunction,` il cui metodo `List getSuccessors(Object state)` restituisce la lista di tutti i possibili successori di *state* applicando tutti gli operatori applicabili
- Attenzione! Non confondete il concetto di successore con un nodo dell'albero di ricerca…

1. http://aima.cs.berkeley.edu/java/doc/javadoc/aima/search/framework/package-tree.html

Riepilogo: fin qui abbiamo…

Se costruisco una classe java che rappresenta uno stato, e che implementa rispettivamente le interfacce `GoalTest` e `SuccessorFunction`, allora posso applicare i seguenti metodi di ricerca:

1. Breadth-first
2. Depth-first
3. Depth-bounded
4. Iterative Deepening

E la strategia a Costo Uniforme?

Se voglio sapere il costo per giungere a tale soluzione, devo anche conoscere il **costo degli operatori** applicati per giungervi…

- A tal scopo è specificata l'interfaccia

`aima.search.framework.StepCostFunction`

con il metodo:

`calculateStepCost(…)`

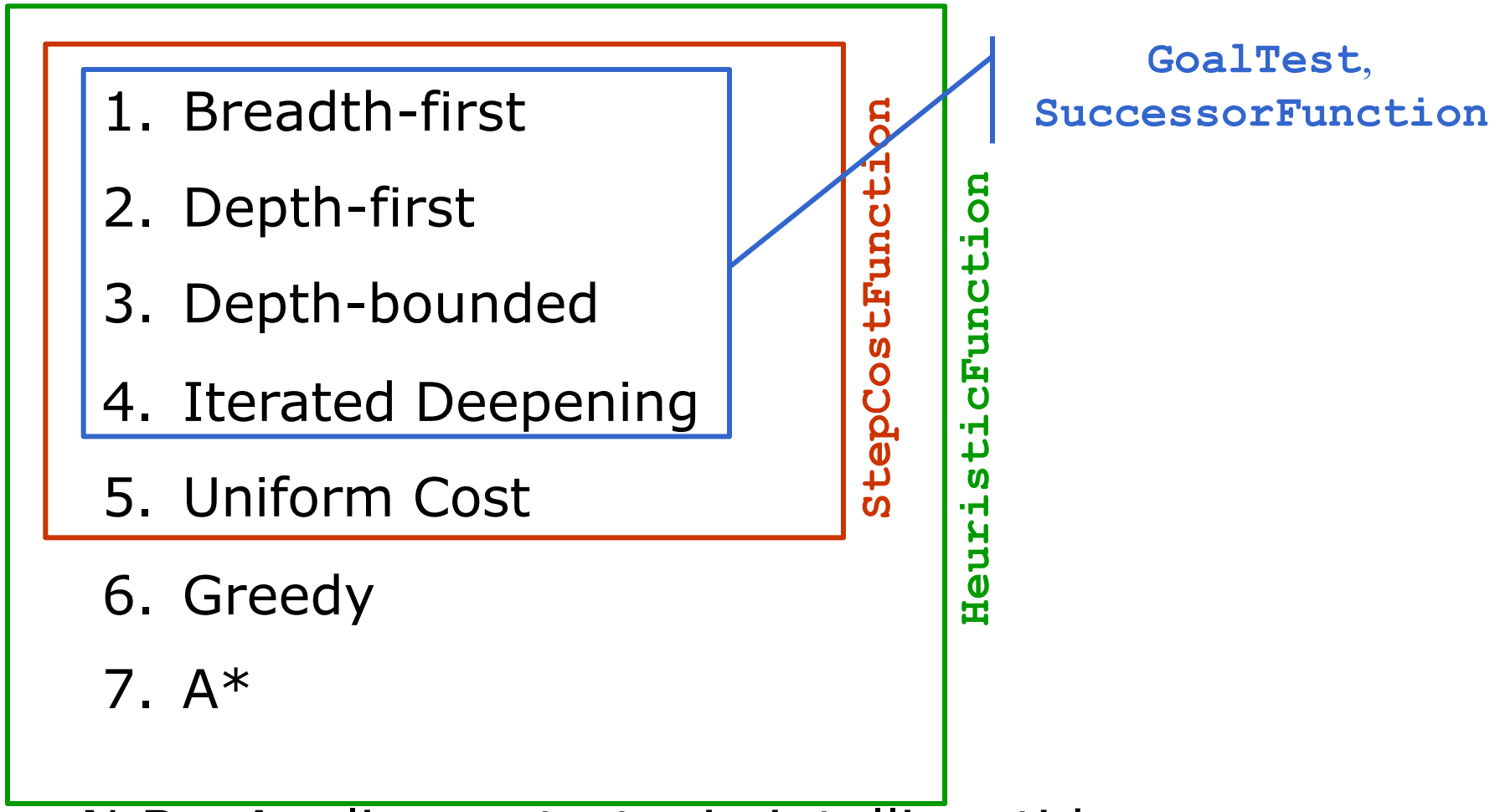
Strategie Informate

- E se voglio applicare delle strategie informate?
- Ricordiamoci la definizione di **euristica**: una funzione che mi restituisce una stima (più o meno esatta) di quanto mi costa giungere fino al goal a partire da un certo stato…
- L'interfaccia `aima.search.framework.HeuristicFunction` definisce il metodo

```
int getHeuristicValue(Object state);
```

- Ricordiamoci che poi, a seconda della strategia, si può scegliere di considerare soltanto l'euristica, o una qualche funzione più elaborata…

Riepilogo: come implementare uno stato



N.B.: Applicare strategie intelligenti ha un costo in termini di "conoscenza" che devo fornire…

Com'è costruita la libreria `aima.search`

Fornisce la classe:

- **`aima.search.framework.Problem`**

  rappresentante il problema, a cui è possibile fornire lo stato iniziale (una qualunque istanza di `java.lang.Object`), e le implementazioni delle interfacce (secondo necessità):

  - `GoalTest`
  - `SuccessorFunction`
  - `StepCostFunction`
  - `HeuristicFunction`

Com'è costruita la libreria `aima.search`

Fornisce direttamente l'implementazione dei nodi di un albero (grafo) di ricerca, tramite le classi:

- **`aima.search.framework.Node`**,
- **`aima.search.framework.QueueSearch`**, che

implementa l'algoritmo GeneralSearch, con le sottoclassi:

- **`BreadthFirstSearch`**
- **`DepthFirstSearch`**
- **`DepthLimitedSearch`**
- **`IterativeDeepeningSearch`**
- **`GreedyBestFirstSearch`**
- **`SimulatedAnnealingSearch`**
- **`HillClimbingSearch`**
- **`AStarSearch`**

package aima.search.uninformed

package aima.search.informed

L'algoritmo generale di ricerca

```
function GENERAL-SEARCH(problem, QUEUING-FN) returns a solution, or failure

    nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
    loop do
        if nodes is empty then return failure
        node ← REMOVE-FRONT(nodes)
        if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
        nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
    end
```

Tramite l'argomento **Queuing-Fn** viene passata una funzione per accodare i nodi ottenuti dall'espansione

Com’è costruita la libreria `aima.search`

Per attuare le varie strategie, la libreria aima.search utilizza diverse queuening function:

- `aima.search.datastructures.FIFOQueue`
- `aima.search.datastructures.LIFOQueue`
- `aima.search.datastructures.PriorityQueue`

Com’è costruita la libreria `aima.search`

Alla classe QueueSearch (cioè alle sue sottoclassi) è possibile specificare anche se si sta effettuando la ricerca su un albero o su un grafo (controlla di non ripassare per uno stato già visitato). A tal scopo fornisce anche le classi:

- `aima.search.framework.TreeSearch`
- `aima.search.framework.GraphSearch`

Com'è costruita la libreria `aima.search`

Viene fornita infine la classe:

`aima.search.framework.SearchAgent`

che provvede direttamente a cercare la soluzione nello spazio degli stati.

Riceve come parametri di ingresso un problema (istanza della classe **`Problem`**) ed una strategia di ricerca (istanza dell'interfaccia **`Search`**).

Risolvere un problema con un algoritmo di ricerca

- Definire un oggetto di tipo `Problem` che riceve come parametro un insieme di oggetti rappresentanti lo stato iniziale, la funzione generatrice di stati successori, lo stato goal (goalTest), eventualmente la funzione costo e, nel caso di ricerca informata, la funzione euristica;
- Definire un oggetto di tipo `Search` come istanza della classe dell'algoritmo che si intende utilizzare (cf. slide 21);
- Definire un oggetto di tipo `SearchAgent` che riceve gli oggetti Problem e Search;
- Le funzioni `printActions/printInstrumentation` stampano la soluzione (cammino) e informazioni statistiche dipendenti dall'algoritmo.

Un primo problema semplice: Missionari e Cannibali

- 3 missionari e 3 cannibali devono attraversare un fiume. C'è una sola barca che può contenere al massimo due persone. Per evitare di essere mangiati i missionari non devono mai essere meno dei cannibali sulla stessa sponda (stati di fallimento).

- Stato: sequenza ordinata di tre numeri che rappresentano il numero di missionari, cannibali e barche sulla **sponda** del fiume **da cui sono partiti**.

- Perciò lo stato iniziale è: (3,3,1) (nota l'importanza dell'astrazione).

Un primo problema semplice: Missionari e Cannibali

- **Operatori**: gli operatori devono portare in barca
  - 1 missionario, 1 cannibale,
  - 2 missionari,
  - 2 cannibali,
  - 1 missionario
  - 1 cannibale.
- Al più 5 operatori (grazie all'astrazione sullo stato scelta).
- Test **Obiettivo**: **Stato finale (0,0,0)**
- Costo di cammino: numero di traversate.
- **Costo traversata**=1.
- *Verificare cosa accade con le diverse strategie breadth first, depth first e depth bounded*

Main

```
MCStato initState = new MCStato(); //crea lo stato
                                   //iniziale (3,3,true)

Problem problem = new Problem(
      initState,
      new
      MCFunzioneSuccessore(),
      new MCStato());

Search search = new BreadthFirstSearch(new TreeSearch());

SearchAgent agent = new SearchAgent(problem, search);


// semplici metodi di stampa dei risultati
printActions(agent.getActions());
printInstrumentation(agent.getInstrumentation());
```

Lo stato

```
public class MCStato implements
             GoalTest,
             StepCostFunction {

//attributi

//getter

//costruttore e costruttore di default a 3 argomenti

//override isGoalState: controlla se lo stato finale
   è stato raggiunto

//override calculateStepCost: restituisce il costo di
   un passo

}
```

Funzione successore

```
public class MCFunzioneSuccessore
                    implements SuccessorFunction {

   public List getSuccessors(Object stato) {

   // creare la lista dei successori (usare ArrayList)
   // recuperare lo stato corrente
   // testare se la mossa è possibile
   // eseguire la mossa (5 possibili)
   // controllare se il nuovo stato raggiunto è
   consentito e aggiungerlo alla lista successori come
   istanza di Successor
   }
   //metodo per verifica stato consentito
   //metodi per le mosse
   // es.: private MCStato muoviMM(MCStato stato) {}
}
```

Funzione successore

```
public class MCFunzioneSuccessore
                    implements SuccessorFunction {

   public List getSuccessors(Object stato) {

   List successori = new ArrayList();
      if (stato instanceof MCStato) {
            MCStato corrente = (MCStato)stato;
            ..........
}

//metodo per verifica stato consentito: restituisce
   true se i missionari non sono in pericolo su
   entrambe le sponde
//metodi per le mosse: controllano dove si trova la
   barca e costruiscono uno nuovo stato
   es.: private MCStato muoviMM(MCStato stato) {}
}
```

utili

```
private static void printInstrumentation(Properties
    properties) {
    Iterator keys = properties.keySet().iterator();
    while (keys.hasNext()) {
            String key = (String) keys.next();
            String property = properties.getProperty(key);
            System.out.println(key + " : " +
property.toString());
    }

}

private static void printActions(List actions) {
    for (int i = 0; i < actions.size(); i++) {
            String action = (String) actions.get(i);
            System.out.println(action);
    }
}
```

Missionari e Cannibali parametrico

- Modificare il programma in modo da gestire un numero generico di missionari e cannibali

- Stato: sequenza ordinata di 5 numeri che rappresentano il numero totale di missionari, il numero di missionari sulla sponda di partenza, il numero totale di cannibali, il numero di cannibali sulla sponda di partenza e la posizione della barca.

- Gli operatori sono gli stessi

In Laboratorio

1. Lanciare Eclipse e creare un nuovo progetto Java
2. Scaricare dalla home page del corso il file *aima.0.95.jar* nella cartella *lib* del progetto (da creare se non esiste), e includerlo nelle librerie del progetto (scheda *Libraries* del *Java Build Path,* accessibile dalle *Properties* del progetto)
3. Creare nuove classi al fine di risolvere i problemi proposti
   - Missionari e Cannibali
   - Quadrato magico
   - U2